Initial Questionnaire – The Results Are In!

- 56% of all students in CS15 have no prior coding experience!
- 81% of all students in CS15 were not yet comfortable using Java at the beginning of the course
- Only 1% have used the JavaFX graphics library (which you will learn about next week)

Review: Constructors

Correct
```
public class SamBot {
    private int _numsteps;
    public SamBot() {
        _numsteps = 5;
    }
}
```

Incorrect
```
public class SamBot {
    private int _numsteps;
    public SamBot() {
        _numsteps = 5;
        new SamBot();
    }
}
```

Incorrect
```
public class SamBot {
    private int _numsteps;
    public SamBot(new SamBot()) {
        _numsteps = 5;
    }
}
```

- Parentheses after constructor, not in class declaration
- The constructor (called by another instance) makes the SamBot - do not "new" an object of the same class in the constructor
- Constructor could take in parameters - but not a new SamBot

Review: Assigning Variables

- What is the output of this program?
  10
  50
- Remember: in Java, = is an operator for variable assignment
  o This is different from the mathematical notion of equality!

Review: Parameter Passing

```
public class NumberPrinter {
    public NumberPrinter() {
        this.printNumbers();
    }
    public void printNumbers() {
        int a = 1;
        int b = 10;
        a = b;
        b = 50;
        System.out.println(a);
        System.out.println(b);
    }
}
```

```
public class DogGroomer {
    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog
    }
}
```

```
public class Dog {
    Dog(String name) {
        this.name = name;
    }
    String name;
}
```

```
public class PetShop {
    /* This is the constructor! */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
    }
}
```

- Calling the groom method requires passing in an existing Dog as argument
- Existing Dog is dogE, and Java will substitute dogE for shaggyDog every place the groom method refers to it
Review: Method Signatures

The same signature can occur in multiple classes (e.g., Cat and Dog) without ambiguity — the type of the instance on which the call will be made lets Java disambiguate.

If a class is a subclass of another, and both have the same method signature(s), then overriding occurs — method resolution disambiguates.

If there is no inheritance relationship between the classes with the same signature, there is no overriding, and the methods each work independently.

This applies to methods in support code classes as well.

Review: Types of Variables

<table>
<thead>
<tr>
<th>Type of Variable:</th>
<th>Instance</th>
<th>Local</th>
<th>No Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>When to use it:</td>
<td>If you need to refer to it throughout the class</td>
<td>If you only need to refer to it inside of a method</td>
<td>When you never need to refer to it in the future after construction within the scope of the class in which it was created (i.e., created and passed as an argument)</td>
</tr>
<tr>
<td>Syntax:</td>
<td>_variable</td>
<td>variable</td>
<td>Somewhere in andy.teach(){</td>
</tr>
<tr>
<td></td>
<td>variable</td>
<td></td>
<td>_andy.takeWater(new H2OBottle());</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Note: The underscore notation is only a CS15 style convention for instance variables — it doesn’t affect how Java interprets your code.

Introduction

- Inheritance review
- What is polymorphism?
- Advantages of polymorphism
- Polymorphism with interfaces
- Limitations of polymorphism
Coding Generically

- We've seen this abstraction pattern over and over so far...
  - **classes and instances**: factor out common capabilities of instances into a class
  - **parameters**: write a generic method that behaves differently based on input
  - **inheritance**: factor out common capabilities of similar classes into superclass
  - **interfaces**: factor out common capabilities of dissimilar classes into interface

- This lecture: **Polymorphism**! Generic coding to the max

Inheritance Review

- **Convertible**, **CS15Mobile**, and **Van** are all subclasses of **Vehicle**
- Subclasses inherit properties and capabilities from their superclass
- Subclass may choose to **override** a method declared in the superclass to further specialize that capability
- Thus, subclasses may all respond to the same method call differently (e.g., move in their own way)

Polymorphism (1/6)

- **Polymorphism** is a fancy word for “multiple forms”
- The big idea: we can **refer to an instance of a subclass as if it were an instance of its superclass**

Polymorphism (2/6)

```java
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;

    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
}
```
Polymorphism (3/6)

- In the constructor of `Racetrack`, we instantiate our `Vehicle`
- `Vehicle` is an abstract class—cannot instantiate it directly
- Instead, we instantiate a `Van`, a `Convertible`, and a `CS15Mobile`, and assign them to `_vehicle1`, `_vehicle2`, and `_vehicle3`
- `Vehicle` thus takes on three different forms: instances of its subclasses

```java
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;
    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
}
```

Polymorphism (4/6)

- In this example, declared type of `_vehicle1`, `_vehicle2`, and `_vehicle3` is `Vehicle`
- But `_vehicle1`'s actual type is `Van`, `_vehicle2`'s actual type is `Convertible`, and `_vehicle3`'s actual type is `CS15Mobile`
- Thus, `_vehicle1` is an instance of `Van`, but we're referring to it as if it were just a generic `Vehicle`

```java
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;
    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
}
```

Polymorphism (5/6)

- Now we've written a `race` method. When called, it tells each `Vehicle` to move
- What happens?
  - `_vehicle1` is a `Van`, so it moves slowly
  - `_vehicle2` is a `Convertible`, so it moves fast
  - `_vehicle3` is a `CS15Mobile`, so it moves at moderate speed

```java
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;
    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
    public void race() {
        _vehicle1.move();
        _vehicle2.move();
        _vehicle3.move();
    }
}
```

Polymorphism (6/6)

- Each object responds to method calls as specified by its actual type—method resolution
- `race` doesn't care what types of `Vehicle` it's dealing with
- If we wanted to race different types of `Vehicle`, we would only have to modify the constructor
- Good generic programming, but not that impressive...
- The real power of polymorphism is still to come!

```java
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;
    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
    public void race() {
        _vehicle1.move();
        _vehicle2.move();
        _vehicle3.move();
    }
}
```
Without Polymorphism (1/2)

- Here's a `Driver` class
- All `Drivers` have the ability to drive a `Convertible`
- But there are other kinds of `Vehicles` out there... We want `Drivers` to be able to drive `Vans` and `CS15Mobiles` too!
- Let's tackle the problem without using polymorphism

```java
public class Driver {
    // constructor elided
    public void driveConvertible(Convertible c) {
        c.move();
    }
}
```

Without Polymorphism (2/2)

- This works! We just wrote a different method for each kind of `Vehicle`
- But what if we want to introduce more kinds of `Vehicle`, like `Trucks` and `SUVs`?
- Would need to keep writing new methods for each type of `Vehicle` we might ever want a `Driver` to drive!

```java
public class Driver {
    // constructor elided
    public void driveConvertible(Convertible c) {
        c.move();
    }
    public void driveCS15Mobile(CS15Mobile cm) {
        cm.move();
    }
    public void driveVan(Van v) {
        v.move();
    }
}
```

With Polymorphism! (1/3)

- This is exactly the problem polymorphism was designed to solve
- Instead of declaring many different methods, each with a parameter of a different declared type, can use single method that takes in any `Vehicle`

```java
public class Driver {
    // constructor elided
    public void drive(Vehicle vehicle) {
        vehicle.move();
    }
}
```

With Polymorphism! (2/3)

- In a generic `drive()`, specify that the declared type of the parameter is `Vehicle`
- Caller will pass in an instance of a specific subclass (like `CS15Mobile`) as argument
- Subclass type is actual type, a subtype of the declared type
- Thus, generic method refers to instance of subclass as if it were an instance of its superclass...if you pass in a `CS15Mobile`, its specific `move()` method will be called – polymorphic parameter passing!
With Polymorphism! (3/3)

- Here’s an example of calling the `drive` method, passing in a different type of `Vehicle` as an argument each time
- Can pass in an instance of any subclass of `Vehicle`
- And method resolution will invoke the method for the subclass!
- Relaxation of ultra-strict type matching

```java
public class DriversEdApp {
    Driver _studentDriver;
    public DriversEdApp() {
        _studentDriver = new Driver();
    }
    public void startDrivingTest() {
        _studentDriver.drive(new Convertible());
        _studentDriver.drive(new CS15Mobile());
        _studentDriver.drive(new Van());
    }
}
```

Polymorphism has to be used carefully...

- Consider the `Zookeeper` class, which has the method `feedLion`
- `feedLion` can take in any `Lion`
- Let’s say we have the following code in another class:
  ```java
  public class Zookeeper {
    public void feedLion(Lion lion) {
      // implementation elided
    }
  }
  ```
- What is the declared type? The actual type?
- Can we pass `cat` into the `feedLion` method?
  - No!

Polymorphism is Tricky...

- `feedLion` will only accept an argument of type `Lion`, and can call any `Lion` methods on its parameter
- `cat` is declared as type `Cat` - as far as Java knows, could be any kind of `Cat` (`Tiger`, `Panther`, etc.), and `Cats` won’t know all `Lion` methods!
- The argument’s declared type must be the same class, or a subclass, of the parameter’s declared type
- What’s an easy fix (if you didn’t actually use any `Lion-specific` methods?)

```java
public class Zookeeper {
    public void feedLion(Lion lion) {
      // implementation elided
    }
}
```

Polymorphism With Interfaces (1/4)

- Polymorphism with interfaces looks much the same as it does with classes
- Imagine that the classes `CS15Mobile` and `Watch` both implement the `Repairable` interface

```java
public interface Repairable {
    public void repair();
}
```
Polymorphism With Interfaces (2/4)

- Let's say we have a `RepairShop` class
- The `RepairShop` has the method `fix`, which takes in any object that implements the interface `Repairable`

```
public class RepairShop {

    public void fix(Repairable brokenThing) {
        brokenThing.repair();
    }
}
```

Polymorphism With Interfaces (3/4)

- Notice that we use the `interface name` as the declared type of the parameter
- Interfaces can be used as types of variables and parameters, just like classes can!

```
public class RepairShop {

    public void fix(Repairable brokenThing) {
        brokenThing.repair();
    }
}
```

Polymorphism With Interfaces (4/4)

- Just like with superclasses and subclasses, object passed in will respond to method call specifically based on its `actual type`
- With polymorphism, interfaces are truly useful--can write a single method to deal with any object that fulfills a certain contract

```
public class RepairShop {

    public void fix(Repairable brokenThing) {
        brokenThing.repair();
    }
}
```

Limitations of Polymorphism (1/2)

- Let's go back to the `Racetrack` example
- We know that `_vehicle2` is actually a `Convertible`
- Let's say that `Convertibles` have a method called `putTopDown`

```
public class Racetrack {

    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;

    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }

    public void race() {
        _vehicle1.move();
        _vehicle2.move();
        _vehicle3.move();
    }
}
```
Limitations of Polymorphism (2/2)

- No!
- Can only call methods that are defined in (or inherited by) the declared type of the object
- We're referring to _vehicle2 as a generic Vehicle - can only tell it to do things that all Vehicles know how to do
- Same premise applies with interfaces - if declared type is an interface, can only call methods defined in that interface

```
public class Racetrack {
    private Vehicle _vehicle1;
    private Vehicle _vehicle2;
    private Vehicle _vehicle3;
    public Racetrack() {
        _vehicle1 = new Van();
        _vehicle2 = new Convertible();
        _vehicle3 = new CS15Mobile();
    }
    public void race() {
        _vehicle1.move();
        _vehicle2.move();
        _vehicle3.move();
    }
}
```

Is This Code Correct? (1/7)

- Let's say that FordVan is a subclass of Van, and both are concrete classes
- Will this code compile?
- No!

```
public class CarDealership {
    private FordVan _myFordVan;
    public CarDealership() {
        _myFordVan = new Van();
    }
}
```

Is This Code Correct? (2/7)

- Polymorphism doesn't work backwards!
- Since FordVan is the declared type of _myFordVan, we should be able to call any of FordVan's methods on it
- A generic Van wouldn't know how to respond to FordVan's specialized methods!
- A FordVan “is a” Van, but a Van is typically not a FordVan

```
public class CarDealership {
    private FordVan _myFordVan;
    public CarDealership() {
        _myFordVan = new Van();
    }
}
```

Is This Code Correct? (3/7)

- Assume that Cat and Bird are both subclasses of Animal
- Will this code compile?
- No!

```
public class Cat extends Animal {
    // constructor elided
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```

```
public class BirdCatchingApp {
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```

```
public class BirdCatchingApp {
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```

```
public class BirdCatchingApp {
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```
Is This Code Correct? (4/7)

- The `Cat`'s `catchBird` method takes in a `Bird` as a parameter.
- But we've declared `tweety` as an `Animal`, meaning we must refer to `tweety` as only an `Animal`.
- `catchBird` is declared to allow calls to `Bird`-specific methods on whatever argument is passed in, but we can only allow methods that are common to all `Animals` to be called on `tweety`.

```java
public class Cat extends Animal {
    // constructor elided
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```

```java
public class BirdCatchingApp {
    public BirdCatchingApp() {
        Cat sylvester = new Cat();
        Animal tweety = new Bird();
        sylvester.catchBird(tweety);
    }
}
```

Is This Code Correct? (5/7)

- We can fix this by just declaring `tweety` as a `Bird`.

```java
public class Cat extends Animal {
    // constructor elided
    public void catchBird(Bird unluckyBird) {
        // implementation elided
    }
}
```

```java
public class BirdCatchingApp {
    public BirdCatchingApp() {
        Cat sylvester = new Cat();
        Bird tweety = new Bird();
        sylvester.catchBird(tweety);
    }
}
```

Is This Code Correct? (6/7)

- Assume that `Mango`, `Apple` and `Banana` are subclasses of `Fruit`.

```java
public class Zoo {
    private Monkey _monkey;
    public Zoo() {
        _monkey = new Monkey();
        Fruit mango = new Mango();
        Apple apple = new Apple();
        _monkey.eat(mango);
        _monkey.eat(apple);
    }
}
```

```java
public class Monkey {
    // constructor elided
    public void eat(Fruit fruit) {
        fruit.beDige sted();
    }
}
```

Is This Code Correct? (7/7)

```java
public class Zoo {
    private Monkey _monkey;
    public Zoo() {
        _monkey = new Monkey();
        Fruit mango = new Mango();
        Apple apple = new Apple();
        _monkey.eat(mango);
        _monkey.eat(apple);
        _monkey.eat(new Banana());
    }
}
```

```java
public class Monkey {
    // constructor elided
    public void eat(Fruit fruit) {
        fruit.beDige sted();
    }
}
```

Yes, this code works!
That’s It!
- Polymorphism is the ultimate tool for coding generically!
  - method takes in a parameter of type <superclass> or <interface> (e.g. Vehicle or Repairable)
  - result: can pass instance of any subclass into that method!
- Limitation: When calling methods on an object, can only ever call methods of its declared type!
  - if myCar is declared as a Vehicle, can’t call putTopDown on it, even if it’s actually a Convertible
- Polymorphism thus lets instances of subclasses and interface-implementing classes become “multiple forms” of the generic parameter or variable

Announcements
- LiteBrite on-time handin is tonight at 11:59PM
  - Late handin is Saturday 10/3 at 10pm
- TA SafeHouse is released today
  - Design questions are due Sunday 10/4 2PM
- Today is the last day to get checked off for lab 1 at TA hours